Specification Matching of Software Components

Amy Moormann Zaremski and Jeannette M. Wing

March 1995

CMU-CS-95-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to *SIGSOFT '95 Foundations of Software Engineering Symposium*

Abstract

Specification matching is a way to compare two software components. In the context of software reuse and library retrieval, it can help determine whether one component can be substituted for another or how one can be modified to fit the requirements of the other. In the context of object-oriented programming, it can help determine when one type is a behavioral subtype of another. In the context of system interoperability, it can help determine whether the interfaces of two components mismatch. We use formal specifications to describe the behavior of software components, and hence, to determine whether two components match. We give precise definitions of not just exact match, but more relevantly, various flavors of relaxed match. These definitions capture the notions of generalization, specialization, substitutability, subtyping, and interoperability of software components. We write our formal specifications of components in terms of pre- and post-condition predicates. Thus, we rely on theorem proving to determine match and mismatch. We give examples from our implementation of specification matching using the Larch Prover.

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government. This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.

Keywords: software libraries, component retrieval, specification matching, subtyping, interoperability, theorem proving

1. Motivation and Introduction

Specification matching is a process of determining if two software components are related. It underlies understanding this seemingly diverse set of questions:

- *Retrieval.* How can I retrieve a component from a software library based on its semantics, rather than syntactic structure?
- *Reuse.* How might I adapt a component from a software library to fit the needs of a given subsystem?
- *Substitution.* When can I replace one software component with another without affecting the observable behavior of the entire system?
- *Subtype.* When is an object of one type a subtype of another?
- *Interoperation.* Why is it so difficult to make two independently developed components work together?

In retrieval, we search for all library components that satisfy a given query. In reuse, we adapt a component to fit its environmental constraints, based on how well the component meets our requirements. In substitution, we expect the behavior of one component to be observably equivalent to the other's; a special case of substitution is when a subtype object is the component substituting for the supertype object. In interoperation, we want one component to interact properly with the other. Common to answering these questions is deciding when one component *matches* another, where "matches" generically stands for "satisfies," "meets," "is equivalent to," or "interacts properly with." Common to these kinds of matches is the need to characterize the dynamic behavior, i.e., semantics, of each software component.

It is rarely the case that we would want one component to match the other "exactly." In retrieval, we want a close match; as in any information retrieval context [Cor, ML94, SM83], we might be willing to sacrifice precision for recall. That is, we would be willing to get some false positives as long as we do not miss any (or too many) true positives. In determining substitutability, we do not need the substituting component to have the exact same behavior as the substituted, only the same behavior relative to the environment that contains it.

In this paper we lay down a foundation for different kinds of semantic matches. We explore not just *exact match* between components, but many flavors of *relaxed match.* To be concrete and to narrow the focus of what match could mean, we make the following assumptions:

- The software components in which we are interested are *functions* (e.g., C routines, Ada procedures, ML functions) and *modules* (roughly speaking, sets of functions) written in some programming language. These components might typically be stored in a program library, shared directory of files, or software repository.
- Associated with each component, C, is a signature, C_{sig}, and a specification of its behavior, C_{spec}.

Whereas signatures describe a component's type information (which is usually statically-checkable), specifications describe the component's dynamic behavior. Specifications more precisely characterize the semantics of a component than just its signature. In this paper, our specifications are formal, i.e., written in a formally defined assertion language.

Given two components, $C = \langle C_{sig}, C_{spec} \rangle$ and $C' = \langle C'_{sig}, C'_{spec} \rangle$, we define a generic component match predicate, *Match*:

Definition: (*Component Match*)

$$\begin{array}{l} Match\colon Component, Component \rightarrow Bool \\ Match(C, C') = match_{sig}(C_{sig}, C'_{sig}) \ \wedge \ match_{spec}(C_{spec}, C'_{spec}) \end{array} \qquad (1)$$

Two components C and C' *match* if 1) their signatures match, given some definition of signature matching ($match_{sig}$), and 2) their specifications match, given some definition of specification match ($match_{spec}$). Although we define match as a conjunction, we can think of signature match as a "filter" that eliminates the obvious non-matches before trying the more expensive specification match.

There are many possible definitions for the signature match predicate, $match_{sig}$, which we thoroughly analyzed in a previous paper [ZW95]. In the remainder of this paper, for $match_{sig}$, we use for functions type equivalence modulo variable renaming ("exact match" in [ZW95]), and for modules, a partial mapping of functions in the modules with exact signature match on the functions ("generalized module match" in [ZW95]).

In this paper, we focus on the specification match predicate, $match_{spec}$. We write pre-/post-condition specifications for each function, where assertions are expressed in a first-order predicate logic. Match between two functions is then determined by some logical relationship, e.g., implication, between the two pre-/post-conditions specifications. We can then modularly[1] define match between two modules in terms of some kind of match between corresponding functions in the modules. Given our choice of formal specifications, we can exploit state-of-the-art theorem proving technology as a way to implement a specification match engine.

Specification match goes a step beyond signature match. For functions, signature match is based entirely on the functions' types, e.g., $int * int \rightarrow int$, and not at all on their behavior. For example, integer addition and subtraction both have the same signature, but completely opposite behavior; the C library routines *strcpy* and *strcat* have the same signature but users would be unhappy if one were substituted for the other. Given a large software library or a large software system, many functions will have identical signatures but very different behavior. For example, in the C math library nearly two-thirds of the functions (31 out of 47) have signature $double \rightarrow double$. Based on signature match alone, we cannot know if we are interoperating with a function properly or know which of a large number of retrieved functions does what we want. Since specification match takes into consideration more knowledge about the components it allows us to increase the precision with which we determine when two components match.

In what follows, we first briefly describe the language with which we write our formal specifications. We define exact and relaxed match for functions (Section 3) and then for modules (Section 4). We discuss in more detail applications of specification match in the software engineering context in Section 5 and our implementation of a specification matcher using the Larch Prover in Section 6. We close with related work and a summary.

2. Larch/ML Specifications

We use Larch/ML [WRZ93], a Larch interface language for the ML programming language, to specify ML functions and ML modules. Larch provides a "two-tiered" approach to specification [GH93]. In one tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties of a program. Each trait introduces *sorts* and *operators* and defines equality between terms composed of

[1] Pun intended.

the operators (and variables of the appropriate sorts). Appendix A shows the *Sequence* trait, which defines operators to generate sequences (*empty* and *insert*), to return the element or sequence resulting from deleting an element from the beginning (or end) (*first* (*last*) and *butFirst* (*butLast*)), and to return the length of a sequence (*length*) or whether a sequence is empty (*isEmpty*).

In the second tier, the specifier writes *interfaces* in a Larch interface language to describe state-dependent effects of a program (see Figure 1). The Larch/ML interface language extends ML by adding specification information in special comments delimited by $(*+ \ldots +*)$. The **using** and **based on** clauses link interfaces to LSL traits by specifying a correspondence between (programming-language specific) types and LSL sorts. The specification for each function begins with a *call pattern* consisting of the function name followed by a pattern for each parameter, optionally followed by an equal sign (=) and a pattern for the result. In ML, patterns are used in binding constructs to associate names to parts of values (*e.g.*, *(x, y)* names x as the first of a pair and y as the second). The **requires** clause specifies the function's pre-condition as a predicate in terms of trait operators and names introduced by the call pattern. Similarly, the **ensures** clause specifies the function's post-condition. If a function does not have an explicit **requires** clause, the default is **requires** *true*.

```
signature Stack = sig
    (*+ using Sequence  +*)
    type α t
      (*+ based on Sequence.E Sequence.S  +*)

    val create : unit → α t
    (*+ create ( ) = s
      ensures s = empty  +*)

    val push : α t * α → α t
    (*+ push (s,e) = s2
      ensures s2 = insert(e,s)  +*)

    val pop : α t → α t
    (*+ pop s = s2
      requires not(isEmpty(s))
      ensures s2 = butFirst(s)  +*)

    val top : α t → α
    (*+ top s = e
      requires not(isEmpty(s))
      ensures e = first(s)  +*)
  end
```

```
signature Queue = sig
    (*+ using Sequence  +*)
    type α t
      (*+ based on Sequence.E Sequence.S  +*)

    val create : unit → α t
    (*+ create ( ) = q
      ensures q = empty  +*)

    val enq : α t * α → α t
    (*+ enq (q,e) = q2
      ensures q2 = insert(e,q)  +*)

    val rest : α t → α t
    (*+ rest q = q2
      requires not(isEmpty(q))
      ensures q2 = butLast(q)  +*)

    val deq : α t → α
    (*+ deq q = e
      requires not(isEmpty(q))
      ensures e = last(q)  +*)
  end
```

Figure 1: Two Larch/ML Specifications

We will use the Larch/ML interface specifications of Figure 1 as the "library" for our examples of specification matching. It contains module specifications for Stack and Queue, specifying the functions *create*, *push*, *pop*, and *top* on stacks, and *create*, *enq*, *deq*, and *rest* on queues. We specify each function's pre-/post-conditions in terms of operators from the *Sequence* trait.

3. Function Matching

For a function specification, S, we denote the pre- and post-condition as S_{pre} and S_{post}, respectively. S_{pred} defines the interpretation of the function's specification as an implication between the two: $S_{pred} = S_{pre} \Rightarrow S_{post}$. Intuitively, this interpretation means that if S_{pre} holds when the function specified by S is called, S_{post} will hold after the function has executed (assuming the function terminates). If S_{pre} does not hold, there are no guarantees about the behavior of the function. This interpretation of a pre- and post-condition specification is the most common and natural for functions in the standard programming model.

For example, for the Stack *top* function in Figure 1, the pre-condition, top_{pre}, is $not(isEmpty(s))$; the post-condition, top_{post}, is $e = first(s)$; and the specification predicate, top_{pred}, is $(not(isEmpty(s))) \Rightarrow (e = first(s))$.

To be consistent in terminology with our signature matching work, we present function specification matching in the context of a retrieval application. Example matches are between a library specification S and a query specification Q. We assume that variables in S and Q have been renamed consistently[2]. For example, if we compare the Stack *pop* function with the Queue *rest* function, we must rename q to s and $q2$ to $s2$. In this section we examine several definitions of the specification match predicate $(match_{spec}(S,Q))$. We characterize definitions as either grouping pre-conditions S_{pre} and Q_{pre} together and post-conditions S_{post} and Q_{post} together, or relating predicates S_{pred} and Q_{pred}. Both of these kinds of matches have a general form.

Definition: (*Generic Pre/Post Match*)

$$match_{pre/post}(S,Q) = (Q_{pre}\ \mathcal{R}_1\ S_{pre})\ \mathcal{R}_2\ (S_{post}\ \mathcal{R}_3\ Q_{post}) \tag{2}$$

Pre/post matches relate the pre-conditions of each component and the post-conditions of each component. The relations $\mathcal{R}_1$ and $\mathcal{R}_3$ are either equivalence ($\Leftrightarrow$) or implication ($\Rightarrow$), but need not be the same. $\mathcal{R}_2$ is usually conjunction ($\wedge$) but may also be implication ($\Rightarrow$). The matches may vary from this form by dropping some of the terms.

Definition: (*Generic Predicate Match*)

$$match_{pred}(S,Q) = S_{pred}\ \mathcal{R}\ Q_{pred} \tag{3}$$

Predicate matches relate the entire specification predicates of the two components, S_{pred} and Q_{pred}. The relation $\mathcal{R}$ is either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or reverse implication ($\Leftarrow$).

It is important to look at both kinds of match. Which kind of match is appropriate may depend on the context in which the match is being used or on the specifications being compared. We present the pre/post matches in Section 3.1 and the predicate matches in Section 3.2. For each, we present a notion of exact match as well as relaxed matches.

3.1. Pre/Post Matches

Pre/post matches on specifications S and Q relate S_{pre} to Q_{pre} and S_{post} to Q_{post}. We consider four kinds of pre/post matches, beginning with the strongest match and progressively weakening the match by either relaxing the relations $\mathcal{R}_1$ and $\mathcal{R}_3$ from $\Leftrightarrow$ to $\Rightarrow$, relaxing $\mathcal{R}_2$ from $\wedge$ to $\Rightarrow$, or dropping one or more terms.

[2]This renaming is easily provided by the signature matcher, and we are assuming that the signatures of S and Q match.

Exact Pre/Post Match

We begin by instantiating both $\mathcal{R}_1$ and $\mathcal{R}_3$ to $\Leftrightarrow$ and $\mathcal{R}_2$ to $\wedge$ in the generic pre/post match of Definition 2. Two function specifications satisfy the *exact pre/post match* if their pre-conditions are equivalent and their post-conditions are equivalent.

Definition: (*Exact Pre/Post Match*)

$$match_{E-pre/post}(S, Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$$

Exact pre/post match is a strict relation, yet two different-looking specifications can still satisfy the match. Consider for example the following query $Q1$, based on the Sequence trait. $Q1$ specifies a function that returns a sequence whose size is 0, one way of specifying a function to create a new sequence.

```
signature Q1 = sig                                                    (Q1)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qCreate : unit → α t
    (*+ qCreate ( ) = s
      ensures length(s) = 0 +*)
  end
```

Exact pre/post match holds for $Q1$ with both the Stack and Queue *create* functions of Figure 1. (The specifications of Stack and Queue *create* are identical except for the name of the return value.)

Let us look in more detail at how $Q1$ would match the Stack *create* specification. Let S be the specification for Stack *create* and $Q1$ be the query specification. $S_{pre} = true$, $S_{post} = (s = empty)$. $Q1_{pre} = true$, $Q1_{post} = (length(s) = 0)$. Since both S_{pre} and $Q1_{pre}$ are *true*, showing $match_{E-pre/post}(S, Q1)$ reduces to proving $S_{post} \Leftrightarrow Q1_{post}$, or $(s = empty) \Leftrightarrow (length(s) = 0)$. The "if" case $((s = empty) \Rightarrow (length(s) = 0))$ follows immediately from the axioms in the *Sequence* trait about *length*. Proving the "only-if" case $((length(s) = 0) \Rightarrow (s = empty))$ requires only basic knowledge about integers and the fact that for any sequence, s, $length(s) \geq 0$, which is provable from the Sequence trait.

Plug-in Match

Equivalence is a strong requirement. For *plug-in match*, we relax both $\mathcal{R}_1$ and $\mathcal{R}_3$ to $\Rightarrow$ and keep $\mathcal{R}_2$ as $\wedge$ in the generic pre/post match. Under plug-in match, Q matches any specification S whose pre-condition is weaker (to allow at least all the conditions that Q allows) and whose post-condition is stronger (to provide a guarantee at least as strong as Q).

Definition: (*Plug-in Match*)

$$match_{plug-in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

Plug-in match captures the notion of being able to "plug-in" S for Q, as illustrated in Figure 2. A specifier writes a query Q saying essentially:

> I need a function such that if Q_{pre} holds before the function executes, then Q_{post} holds after it executes (assuming the function terminates).

With plug-in match, if Q_{pre} holds (the assumption made by the specifier) then S_{pre} holds (because of the first conjunct of plug-in match). Since we interpret S to guarantee that $S_{pre} \Rightarrow S_{post}$, we can assume that

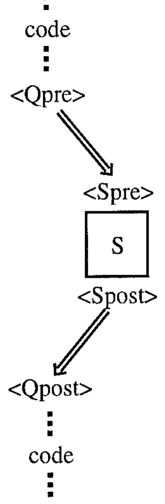

Figure 2: Idea Behind Plug-in Match

S_{post} will hold after executing the plugged-in S. Finally, since $S_{post} \Rightarrow Q_{post}$ from the second conjunct of plug-in match, we are assured of the guarantee the specifier desired.

For example, consider the following query for an insert function:

```
signature Q2 = sig                                                    (Q2)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qEnq : α t * α → α t
    (*+ qEnq (q1, e) = q2
      requires length(q1) < 50
      ensures length(q2) = (length(q1) +1) +*)
  end
```

This query specification requires that an input sequence has fewer than 50 elements, and guarantees that the resulting sequence is one element longer than the input sequence. This is a fairly weak specification. $Q2$ does not satisfy exact pre/post match with any function in the library, but plug-in match holds for $Q2$ with both the Stack *push* and the Queue *enq* functions. Since *push* and *enq* are identical except for their names and the names of the variables, the proof of the match is the same for both.

The pre-condition requirement, $Q_{pre} \Rightarrow S_{pre}$, holds, since $S_{pre} = true$. To show that $S_{post} \Rightarrow Q_{post}$, we assume S_{post} $(q2 = insert(e, q))$, and try to show Q_{post} $(length(q2) = length(q) + 1)$. Substituting for $q2$ in Q_{post}, we have $length(insert(e, q)) = length(q) + 1$, which follows immediately from the equations for *length*.

Plug-in Post Match

Often we are concerned with only the effects of functions, thus a useful relaxation of the plug-in match is to consider only the post-condition part of the conjunction. Most pre-conditions could be satisfied by adding an additional check before calling the function. *Plug-in post match* is also an instance of generic pre/post match, with $\mathcal{R}_3$ instantiated to $\Rightarrow$ but dropping Q_{pre} and S_{pre}.

Definition: (*Plug-in Post Match*)

$$match_{plug-in-post}(S, Q) = (S_{post} \Rightarrow Q_{post})$$

Consider the following query. $Q3$ is identical to Stack *top* except that $Q3$ has no **requires** clause.

```
signature Q3 = sig                                                    (Q3)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qTop : α t → α
    (*+ qTop s = e
      ensures e = first(s) +*)
  end
```

$Q3$ does not satisfy exact pre/post or plug-in match with Stack *top* since $Q3$'s pre-condition is weaker than Stack *top*'s. Since the post-conditions are equivalent, $Q3$ does satisfy plug-in post match with Stack *top*.

Weak Post Match

Finally, consider this even weaker match, *weak post match.* We instantiate $\mathcal{R}_3$ to $\Rightarrow$, as with the plug-in matches, but relax $\mathcal{R}_2$ to $\Rightarrow$ and drop Q_{pre}.

Definition: (*Weak Post Match*)

$$match_{weak-post}(S, Q) = S_{pre} \Rightarrow (S_{post} \Rightarrow Q_{post})$$

A more intuitive, equivalent, predicate is $(S_{pre} \wedge S_{post}) \Rightarrow Q_{post}$. Sometimes assuming the pre-condition of S helps in proving the relationship between S_{post} and Q_{post}. We use S_{pre} and not Q_{pre} since S_{pre} is likely to be necessary to limit the conditions under which we try to prove $S_{post} \Rightarrow Q_{post}$. The additional assumption also means that we will have to provide an additional "wrapper" in our code to guarantee S_{pre} before we call the function specified by S.

For example, suppose we wish to find a function to delete from a sequence using the following query $Q4$:

```
signature Q4 = sig                                                    (Q4)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qRest : α t → α t
    (*+ qRest s = s2
      ensures length(s2) = (length(s) −1) +*)
  end
```

$Q4$ describes a function that returns a sequence whose size is one less than the size of the input sequence. This is a fairly weak way of describing deletion, since it does not specify which element is removed.[3] While intuitively, it would seem related to Stack *pop*, neither plug-in nor plug-in post match holds, because we cannot prove $S_{post} \Rightarrow Q_{post}$ (i.e., $(s2 = butFirst(s)) \Rightarrow (length(s2) = length(s) - 1)$) for the case where

[3] But it still gives us a big gain in precision over signature matching; $Q4$ would not match other functions with the signature $\alpha\ t \rightarrow \alpha\ t$, for example, a function that reverses or sorts the elements in the sequence, or removes duplicates.

$s = empty$. By adding the assumption S_{pre} ($not(isEmpty(s))$), we are able to complete the proof, as we see in the following proof sketch.

Assume $not(isEmpty(s))$	Assume S_{pre}	(1)
Assume $s2 = butFirst(s)$	Assume S_{post}	(2)
$length(s2) = length(s) - 1$	Attempt to prove Q_{post}	(3)
$length(butFirst(s)) = length(s) - 1$	Apply (2) to (3)	(4)
Let $s = insert(ec, sc)$	Since s is not empty (1), and s generated by empty and insert	(5)
$length(butFirst(insert(ec, sc))) = length(insert(ec, sc)) - 1$	Substitute (5) for s in (4)	(6)
$length(sc) = length(insert(ec, sc)) - 1$	Axioms for *butFirst*	(7)
$length(sc) = (length(sc) + 1) - 1$	Axioms for *length*	(8)
$length(sc) = length(sc)$	Axioms for $+, -$	(9)

3.2. Predicate Matches

Recall the generic predicate match (Definition 3):

$$match_{pred}(S, Q) = S_{pred} \; \mathcal{R} \; Q_{pred}$$

where the relation $\mathcal{R}$ is either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or reverse implication ($\Leftarrow$).

Note that this general form allows alternative definitions of the specification predicates. One alternative is $S_{pred} = S_{pre} \wedge S_{post}$, which is stronger than $S_{pred} = S_{pre} \Rightarrow S_{post}$. This interpretation is reasonable in the context of state machines, where the pre-condition serves as a guard so that a state transition occurs only if the pre-condition holds.

As we did with the generic pre/post match, we consider instantiations of the generic predicate match beginning with the strictest.

Exact Predicate Match

We begin with *exact predicate match.* Two function specifications match exactly if their predicates are logically equivalent (i.e., $\mathcal{R}$ is instantiated to $\Leftrightarrow$). This is less strict than exact pre/post match (Section 3.1), since there can be some interaction between the pre- and post-conditions.

Definition: (*Exact Predicate Match*)

$$match_{E-pred}(S, Q) = S_{pred} \Leftrightarrow Q_{pred}$$

Our example $Q1$ still matches with Stack and Queue *create*. In fact, in cases where $S_{pre} = Q_{pre} = true$, the exact pre/post and exact predicate matches are equivalent.

Generalized Match

For generalized match, we relax $\mathcal{R}$ in the generic predicate match to $\Rightarrow$. Generalized match is an intuitive match in the context of queries and libraries: specifications of library functions will be detailed, describing the behavior of the functions completely, but we would like to be able to write simple queries that focus only on the aspect of the behavior that we are most interested in or that we think is most likely to differentiate among functions in the library. Generalized match allows the library specification to be stronger (more general) than the query. Note that generalized match is a weaker match than plug-in match. Also, if we drop the pre-conditions in generalized match, we get plug-in post match.

Definition: (*Generalized Match*)

$$match_{gen}(S, Q) = S_{pred} \Rightarrow Q_{pred}$$

For example, consider the following query, which is the same as $Q4$ but with a **requires** clause.

```
signature Q5 = sig                                                    (Q5)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qRest : α t → α t
    (*+ qRest s = s2
      requires not(isEmpty(s))
      ensures length(s2) = (length(s) −1) +*)
  end
```

Using the exact predicate match, neither the Stack *pop* nor the Queue *rest* specifications satisfy this query. Plug-in match does not work either because we need to assume Q_{pre} $(not(isEmpty(s)))$ to show $S_{post} \Rightarrow Q_{post}$. However, the generalized match with $Q5$ does hold for both of these. The proof is very similar to that for $Q4$ in the weak post match.

Consider another example specifying a function that removes the most recently inserted element of a sequence. This query does not require that the specifier knows the axiomatization of sequences, since the query uses only the sequence constructor, *insert*. The post-condition specifies that the input sequence, s, is the result of inserting an element ee into another sequence ss, and that the element returned, e, is the most recently inserted element (ee). The existential quantifier (**there exists**) is a way of being able to name ee and ss.

```
signature Q6 = sig                                                    (Q6)
    (*+ using Sequence +*)
    type α t (*+ based on Sequence.E Sequence.S +*)
    val qTop : α t → α
    (*+ qTop s = e
      requires not(isEmpty(s))
      ensures there exists ee:Sequence.E, ss:Sequence.S
          ((s = insert(ee, ss)) and (e = ee)) +*)
  end
```

Again, neither the exact nor plug-in matches holds. Generalized match holds for the query with the Stack *top* function, but not Queue *deq*, since the query specifies that the most recently inserted element is returned. To show the generalized match, we consider two cases: $s = empty$, and $s = insert(ec,sc)$. In the first case, the pre-condition for both *top* and *qTop* are false, and thus the match predicate is vacuously true. In the second case, the pre-conditions are both true, and so we need to prove that $S_{post} \Rightarrow Q_{post}$. If we instantiate ee to ec and ss to sc, the proof goes through.

Specialized Match

For *specialized match*, we instantiate $\mathcal{R}$ in the generic predicate match to $\Leftarrow$. Specialized match is the converse of generalized match: $match_{spcl}(S, Q) = match_{gen}(Q, S)$. A function whose specification is weaker than the query might still be of interest as a base from which to implement the desired function. Specialized match allows the library specification to be weaker than the query.

Definition: (*Specialized Match*)

$$match_{spcl}(S, Q) = Q_{pred} \Rightarrow S_{pred}$$

Consider again the query $Q3$, which is the same as Stack *top* but without the pre-condition. Stack *top* is thus weaker than $Q3$, but we can show that $Q3$ implies Stack *top* and hence specialized match holds.

3.3. Relating the Function Matches

$$\begin{tikzcd}
 & & & \text{Exact Pre/Post} \arrow[dll] \arrow[dr] & & \\
 & \text{Exact Predicate} \arrow[dl] \arrow[dr] & & & \text{Plug-in} \arrow[dll] \arrow[dr] & \\
\text{Specialized} \arrow[ddrr] & & \text{Generalized} \arrow[dd] & & & \text{Plug-in Post} \arrow[dl] \\
 & & & & \text{Weak Post} \arrow[dll] & \\
 & & \text{True} & & &
\end{tikzcd}$$

Name of match	predicate symbol	kind of match
Exact Pre/Post	$match_{E-pre/post}$	pre/post
Plug-in Match	$match_{plug-in}$	pre/post
Plug-in Post Match	$match_{plug-in-post}$	pre/post
Weak Post Match	$match_{weak-post}$	pre/post
Exact Predicate Match	$match_{E-pred}$	predicate
Generalized	$match_{gen}$	predicate
Specialized	$match_{spcl}$	predicate

Figure 3: Lattice of Main Function Specification Matches

We relate all our function specification match definitions in a lattice (Figure 3). An arrow from a match $M1$ to another match $M2$ indicates that $M1$ is stronger than $M2$ ($M1 \Rightarrow M2$). We also say that $M2$ is more relaxed than $M1$. The rightmost path in the lattice shows the pre/post matches; the remainder of the matches are predicate matches.

The chart in Figure 3 summarizes the matches we have presented in this section, along with their predicate symbols and whether the match is an instance of the generic pre/post match or the generic predicate match.

Query	$match_{E-pre/post}$	$match_{plug-in}$	$match_{gen}$	$match_{weak-post}$
Q1	Queue *create* Stack *create*	Queue *create* Stack *create*	Queue *create* Stack *create*	Queue *create* Stack *create*
Q2	— —	Queue *enq* Stack *push*	Queue *enq* Stack *push*	Queue *enq* Stack *push*
Q3	—	—	—	Stack *pop*
Q4	—	—	—	Stack *top*
Q5	— —	— —	Queue *rest* Stack *pop*	Queue *rest* Stack *pop*
Q6	—	—	Stack *top*	Stack *top*

Table 1: Which Ones Match What

Table 1 summarizes which of the library functions match each of the example queries for four of the matches we have defined (Exact Pre/Post, Plug-in, Generalized, Weak Post).

4. Module Matching

Function matching addresses the problem of matching particular functions. However, a programmer may need to compare collections of functions, e.g,. ones that provide a set of operations on an abstract data type. Most modern programming language explicitly support the definition of abstract data types through a separate modules facility, e.g., Ada packages, or C++ classes. Modules are also often used just to group a set of related functions, like I/O routines. This section addresses the problem of matching module specifications.

A specification of a module is an interface, $\mathcal{I} = \langle \mathcal{I}_T, \mathcal{I}_F \rangle$, where $\mathcal{I}_T$ is a multiset of user-defined types and $\mathcal{I}_F$ is a multiset of function specifications. For a library interface, $\mathcal{I}_L = \langle \mathcal{I}_{LT}, \mathcal{I}_{LF} \rangle$, to match a query interface, $\mathcal{I}_Q = \langle \mathcal{I}_{QT}, \mathcal{I}_{QF} \rangle$, there must be correspondences both between $\mathcal{I}_{LT}$ and $\mathcal{I}_{QT}$ and between $\mathcal{I}_{LF}$ and $\mathcal{I}_{QF}$. These correspondences vary for exact and relaxed module match.

4.1. Exact Match

Definition: (*Exact Module Match*)

$$\begin{aligned} M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q, match_{fn}) \quad = \quad & \exists \text{ a total function } U_F : \mathcal{I}_{QF} \rightarrow \mathcal{I}_{LF} \text{ such that} \\ & U_F \text{ is one-to-one and onto, and} \\ & \forall\, Q \in \mathcal{I}_{QF}, match_{fn}(U_F(Q), Q) \end{aligned}$$

U_F maps each query function specification Q to a corresponding library function specification, $U_F(Q)$. Since U_F is one-to-one and onto, the number of functions in the two interfaces must be the same (i.e., $|\mathcal{I}_{LF}| = |\mathcal{I}_{QF}|$). The correspondence between each Q and $U_F(Q)$ is that they satisfy the function match, $match_{fn}$. The match parameter ($match_{fn}$) gives us a great deal of flexibility, allowing any of the function matches defined in Section 3 to be used in matching the individual function specifications in a module interface.

4.2. Generalized Match

Should a querier really have to specify all the functions provided in a module in order to find the module? A more reasonable alternative is to allow the querier to specify a set of exactly the functions of interest and match a module that is more *general* in the sense that its set of functions may properly contain the query's set.

Definition: (*Generalized Module Match*)

> $M\text{-}match_{gen}(\mathcal{I}_L, \mathcal{I}_Q, match_{fn})$ is the same as $M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q, match_{fn})$ except U_F need not be onto.

Thus whereas with $M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q, match_{fn})$, $|\mathcal{I}_{LF}| = |\mathcal{I}_{QF}|$, with $M\text{-}match_{gen}(\mathcal{I}_L, \mathcal{I}_Q, match_{fn})$, $|\mathcal{I}_{LF}| \geq |\mathcal{I}_{QF}|$, and $\mathcal{I}_{LF} \supseteq \mathcal{I}_{QF}$ under the appropriate renamings.

What these definitions make clear in a concise and precise manner is the orthogonality between function match and module match. In fact, the module match definitions are completely independent of the fact that we are matching specifications at the function level. If we use the same definitions of module matching, but instantiate $match_{fn}$ with a function signature match, we have module signature matching [ZW95].

5. Applications

As mentioned in Section 1, any problem that involves comparing the behavior of two software components is a potential candidate for specification matching. We examine three such problems: retrieval for reuse, substitution for subtyping, and determining interoperability.

5.1. Retrieval for Reuse

If we have a library of components with specifications, we can use specification matching to retrieve components from the library. Formally, we define the retrieval problem as follows:

Definition: (*Retrieval*)

> *Retrieve*: Query Specification, Match Predicate, Component Library $\rightarrow$ Set of Components
> $Retrieve(Q, match_{spec}, L) = \{C \in L : match_{spec}(C, Q)\}$

Given a query specification Q, a specification match predicate $match_{spec}$, and a library of component specifications L, *Retrieve* returns the set of components in L that match with Q under the match predicate $match_{spec}$. Note that the components can be either functions or modules, provided that $match_{spec}$ is instantiated with the appropriate match. Parameterizing the definition by $match_{spec}$ also gives the user the flexibility to choose the degree of relaxation in the specification match.

Using specification match as part of the retrieval process (or separately on a given pair of components) gives us assurances about how appropriate a component is for reuse. At the function level especially, the various specification matches give us various assurances about the behavior of a component we would like to use. We treat Q as the "standard" we expect a component to meet, and S as the library component we would like to reuse. If exact pre/post match holds on S and Q, we know that S and Q are behaviorally equivalent under all conditions; using S for Q should be transparent. If the exact predicate or plug-in match holds, we know that S can be substituted for Q and the behavior specified by Q will still hold, although we are not guaranteed the same behavior when Q_{pre} is false. If the weak post match holds, we know that the specified behavior holds when S_{pre} is satisfied, which we may be able to guarantee given the specific context in which we use that component.

5.2. Subtyping

Liskov remarked in her OOPSLA '87 keynote address:

> The intuitive idea of a *subtype* is one whose objects provide all the behavior of objects of another type (the *supertype*) plus something extra. What is wanted here is something like the following substitution property [Lea89]: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T. [Lis87].

Behavioral notions of subtyping that attempt to capture this substitutability property have since been defined by many, including America [Ame91], Leavens and his colleagues [Lea89, LW90, DL92], Meyer [Mey88], and Liskov and Wing [LW94]. There are subtle differences between all these subtype definitions, but common to all is the use of pre-/post-condition specifications (1) to describe the behavior of types and (2) to determine whether one type is a subtype of another. Let m_T be a method of supertype T, and m_S be the corresponding method of subtype S. Then America, for example, defines subtype in terms of the following pre-/post-condition rules[4] for each method of the supertype:

- *Pre-condition rule.* $m_T.pre \Rightarrow m_S.pre$.
- *Post-condition rule.* $m_S.post \Rightarrow m_T.post$

which is just our plug-in match. Further, subtyping requires that each method in the supertype T have a corresponding method in the subtype S, but there may be additional methods in S. This corresponds exactly to our generalized module match. More formally,

Definition: *(America) Subtype*

Subtype: Type, Type $\rightarrow$ Bool
Subtype$(S, T) =$ *M-match*$_{gen}(S_{spec}, T_{spec}, match_{plug-in})$

The definitions of subtype suggested by the other researchers can also be cast in terms of specification match in a straightforward way where either or both of *M-match*$_{gen}$ and $match_{plug-in}$ is appropriately changed. In short, the behavioral notion of subtyping is just an instance of our more general notion of specification match.

[4] We omit the abstraction function for simplicity.

5.3. Interoperability

A report on the National Information Infrastructure (NII) states:

> Interoperability is the ability to combine two or more systems into a single acceptably seamless and acceptably efficient system [VLP94].

and argues that demand for interoperability of independently developed systems will grow on an unprecedented scale, in terms of sheer volume, heterogeneity, and complexity of individual systems.

The heart of an interoperability problem is that the interfaces of the two or more systems do not match. Specification match is a way of determining whether two system interfaces match and hence whether the systems can interoperate. We can also learn something about components and their relationships when a match does not occur, i.e., when there is a mismatch. It might be possible to resolve mismatches between two components if we know why they do not match, the more typical scenario when trying to interoperate heterogeneous components.

Suppose we have two components, C and S, that agree to communicate using a remote procedure call protocol. The client C wants to use a service, *op*, provided by S. To interoperate with S, C must at least match the signature of *op* (passing in the right number and types of arguments) and its specification (e.g., establish *op*'s pre-condition).

Even if their signatures and pre-/post-condition specifications match, however, components may still not interoperate. For example, suppose we do not assume that C and S agree on which protocol to use to communicate with each other. If C wants to communicate using non-blocking *send*, but S wants to communicate through remote procedure call (alternating blocking *receive*s and *send*s), then a "protocol mismatch" can occur. For a protocol match, we might require that each one of C's *send*s "lines up" with each one of S's *receive*s and vice versa. However, using CSP-like notation to specify C's and S's protocols, we have:

$$C = send \rightarrow (receive \rightarrow C | send \rightarrow C)$$
$$S = receive \rightarrow send \rightarrow S$$

C might do four *send*s in a row and then do a *receive*; meanwhile, S deadlocks after doing its first *receive* since it wants to do a *send* next, corresponding to a *receive* by C, but conflicting with C's second *send*. That is, the following message sequences do not match:

(C)	*send*	*send*	*send*	*send*	*receive*
(S)	*receive*	*send*	*receive*	*send*	*receive*

If a protocol specification is included in a component's interface specification (i.e., not just signature information and pre-/post-conditions), then we can use a richer notion of specification match to detect this kind of protocol mismatch. We simply extend our notion of match to include additional sub-match predicates, e.g., $match_{protocol}$:

Definition: (*Interoperates*)

Interoperates: Component, Component $\rightarrow$ Bool
$Interoperates(C, C') = Match(C, C') \wedge match_{protocol}(C_{protocol}, C'_{protocol})$

where *Match* is from Definition 1 of Section 1.

6. Implementation

Each of the examples given in this paper have been specified in Larch/ML, translated automatically to LP input, and proven using LP.

For each specification file (e.g., `Stack.sig`), we check the syntax of the specification and then translate it into a form acceptable to LP. Namely, we generate a corresponding `.lp` file (e.g., `Stack.lp`), which contains the appropriate declarations of variables and operators and assertions (axioms) for the pre- and post-conditions of each function specified. Each function *foo* generates two operators, *fooPre* and *fooPost*; the axioms for *fooPre* and *fooPost* are the body of the **requires** and **ensures** clauses of *foo*. Appendix B shows `Stack.lp` and `Q2.lp`, the result of translating the Stack specification from our sample library and the query $Q2$ into LP format.

```
%% PlugIn-Q2-Stack.lp
thaw Stack
thaw Q2
prove (qEnqPre(s, e, s2) => pushPre) /\ (pushPost(s, e, s2) => qEnqPost(s, e, s2))
```

Figure 4: LP input for plug-in match of Stack *push* with $Q2$

We also generate the appropriate LP input to show a given match between two functions. For example, Figure 4 shows the LP input to prove the plug-in match between the Stack *push* function and query $Q2$. The **thaw** *Stack* command loads the state resulting from executing the commands in Stack.lp.

```
%% Gen-Q6-Stack.lp
thaw Stack
thaw Q6
prove (topPre(s, e) => topPost(s, e)) => (qTopPre(s, e) => qTopPost(s, e))
%% additional user input
  resume by induction
    resume by specializing ss to sc
```

Figure 5: LP input for generalized match of Stack *pop* with $Q6$

Since LP is designed as a proof assistant, rather than an automatic theorem prover, some of the proofs require user assistance. The example shown in Figure 4 does not require any assistance from the user. Executing the statements in Figure 4 results ultimately in the response from LP: `[] conjecture`, indicating that LP successfully proved the match conjecture. Generalized match of Stack *pop* with $Q6$ requires some assistance to tell the prover to use induction in the proof, and then how to instantiate the existential variables (Figure 5). Figure 6 shows LP's output script of this proof execution.

7. Related Work

Other work on specification matching has focused on using a particular match definition for retrieval of software components (usually functions). Rollins and Wing proposed the idea of function specification matching and implemented a prototype system in λProlog using plug-in match [RW91]. λProlog does not use equational reasoning, and so the search may miss some functions that match a query but require the use of equational reasoning to determine that they match. The VCR retrieval system [FKS94] uses plug-in match

```
%% exec M-Gen-Q6-Stack
thaw Stack
thaw Q6
prove (topPre(s, e) => topPost(s, e)) => (qTopPre(s, e) => qTopPost(s, e))
  resume by induction
    <> basis subgoal
    [] basis subgoal
    <> induction subgoal
    resume by specializing ss to sc
      <> specialization subgoal
      [] specialization subgoal
    [] induction subgoal
  [] conjecture
  %% End of input from file '/usr/amy/examples/Gen-Q6-Stack.lp'.
%% quit
```

Figure 6: LP output for generalized match of Stack *pop* with $Q6$

with VDM as the specification language. The focus of this work is on efficiency of proving match; the tool performs a series of filtering steps before doing all-out match (e.g., a very relaxed signature matching and model checking). Perry's Inscape system [Per89] is a specification-based software development environment. Its Inquire tool provides predicate-based retrieval in Inscape. Match is either exact pre/post or a form of generalized match. The prototype system has a simplified and hence fairly limited inference mechanism. Also, since specifications must already be provided for software development in Inscape, the user need not write a separate query specification. Jeng and Cheng [JC92] use order-sorted predicate logic specifications. Their match is similar to our generalized function match, but has the additional property that it generates a series of substitutions to apply to the library component to reuse in the desired context. Mili, Mili and Mittermeir [MMM94] define a specification as a binary relation. Specification match is based on the *refines* ordering on relations, somewhat like our generalized match. The PARIS system [KRT87] maintains a library of partially interpreted *schemas*. Each schema includes a specification of restrictions on input to the schema, assertions about how the abstract parts of the schema can be instantiated, and assertions about the results of the schema. Matching corresponds to determining whether a partial library schema could be instantiated to satisfy a query. The system does some reasoning about the schemas but with a limited logic. Katoh, Yoshida and Sugimoto [KYS85] use English-like specifications and queries that are translated into first-order predicate logic formulas. They use "ordered linear resolution" to determine matching between a query and specification, and include relaxations for changing the order of parameters, making some parameters constants, or renaming subroutines. However, the match does not verify that the subroutines match and checks only for equivalence, not permitting any inference.

To summarize, our work on specification matching is more general than the above in three ways: We handle not just function match, but module match; we have a framework, which is extremely modular (e.g., function match is a parameter to module match; specification match is one conjunct of component match), within which we can express each of the specific matches "hardwired" in the definitions used by others; and (3) we have flexible prototype tool that lets us easily experiment with all the different matches. Finally, we are not wedded to just the application of software retrieval; we see the need to understand specification match as it relates to other application areas.

Signature matching can be viewed as a very restricted form of specification matching. Work in this area has focused on taking advantage of the expressiveness and theoretical properties of type systems to define various forms of relaxed matches [ZW95, DC92, Rit92, RT89, SC94].

Less closely related work, but relevant to our context of software library retrieval, divides into two categories: text-based information retrieval [AS86, PD89, MBK91] and AI-based semantic net classifications [OHPDB92, FHR91]. The advantage to these approaches is that many efficient tools are available to do the search and match in these structures. The disadvantage is that the characterization of the component's behavior is completely informal.

8. Summary

This paper makes three specific contributions with respect to specification matching: foundational definitions, descriptions of applications, and a report on a prototype tool.

By providing precise definitions, this paper lays the groundwork for understanding when two different software components are related, in particular when their specifications match. Though we consider in detail functions and modules, exact and relaxed match, and formal pre-/post-condition specifications, the general idea behind specification matching is to exploit as much information associated with the description of software components as possible.

Though our notion of specification match was originally motivated by the software library retrieval application, it is more generally applicable to other areas of software engineering, for example, determining subtyping in designing class hierarchies, or detecting an interoperability problem in a heterogeneous distributed system.

Finally, by building a working specification match engine, we demonstrated the feasibility of our ideas. By providing the community with a tool, we are now in the position to explore their pragmatic implications.

References

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *LNCS*, pages 60–90. Springer-Verlag, NY, 1991.

[AS86] Susan P. Arnold and Stephen L. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. Technical Report 86-CSE-22, Southern Methodist University, October 1986.

[Cor] InfoSeek Corporation. Infoseek home page. Santa Clara, California. http://www.infoseek.com.

[DC92] Roberto Di Cosmo. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 200–210, January 1992.

[DL92] Krishna Kishore Dhara and Gary T. Leavens. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, November 1992.

[FHR91] Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering*, May 1991.

[FKS94] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-based software component retrieval tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[JC92] J.-J. Jeng and B. H. C. Cheng. Formal methods applied to reuse. In *Proceedings of the* 5^{th} *Workshop in Software Reuse*, 1992.

[KRT87] Shmuel Katz, Charles A. Richter, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the* 9^{th} *International Conference on Software Engineering*, pages 377–385, March 1987.

[KYS85] Hideki Katoh, Hiroyuki Yoshida, and Masakatsu Sugimoto. Logic-based retrieval and reuse of software. Technical Report TR-153, Institute for New Generation Computer Technology, October 1985.

[Lea89] Gary Leavens. Verifying object-oriented prograsm that use subtypes. Technical Report 439, MIT Laboratory for Computer Science, February 1989. Ph.D. thesis.

[Lis87] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87: Addendum to the Proceedings*, pages 17–34, 1987.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA '90 Proceedings*, 1990.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[MBK91] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 8(17):800–813, August 1991.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[ML94] M. Mauldin and J. Leavitt. Web-agent related research at the CMT. In *ACM Special Interest Group on Networked Information Discovery and Retrieval (SIGNIDR-94)*, August 1994.

[MMM94] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement-based approach. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*, May 1994.

[OHPDB92] Eduardo Ostertag, James Hendler, Rubén Prieto-Díaz, and Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, July 1992.

[PD89] Rubén Prieto-Díaz. Classification of reusable modules. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Vol. 1: Concepts and Models*, pages 99–123. ACM Press, N.Y., 1989.

[Per89] Dewayne E. Perry. The Inscape environment. In *Proceedings of the* 11^{th} *International Conference on Software Engineering*, pages 2–12, 1989.

[Rit92] Mikael Rittri. Retrieving library identifiers via equational matching of types. Technical Report 65, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, January 1990 (reprinted with corrections May 1992).

[RT89] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Conference on Functional Programming Languages and Computer Architectures*, pages 166–173, September 1989.

[RW91] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming*, June 1991.

[SC94] David W.J. Stringer-Calvert. Signature matching for Ada software reuse. Master's thesis, University of York, 1994.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[VLP94] Mary Vernon, Edward Lazowsk, and Stewart Personick, editors. *R&D for the NII: Technical Challenges.* Interuniversity Communications Council, Inc. (EDUCOM), 1994.

[WRZ93] J.M. Wing, E. Rollins, and A. Moormann Zaremski. Thoughts on a Larch/ML and a new application for LP. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch.* Springer Verlag, 1993.

[ZW95] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching, a Tool for Using Software Libraries. To appear, *ACM Transactions on Software Engineering and Methodology*, 1995. An earlier version appeared in *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, December 1993.

A The Sequence Trait

The *Sequence* trait defines operators to generate sequences (*empty* and *insert*), to return the element or sequence resulting from deleting an element from the beginning (or end) (*first* (*last*) and *butFirst* (*butLast*)), and to return the length of a sequence (*length*) or whether a sequence is empty (*isEmpty*).

```
Sequence(E, S) : trait
    includes Integer
    introduces
        empty :→ S
        insert : E, S → S
        first : S → E
        last : S → E
        butFirst : S → S
        butLast : S → S
        isEmpty : S → Bool
        length : S → Int

    asserts
        S generated by empty, insert
        S partitioned by isEmpty, length
        ∀ e : E, s : S
            first(insert(e, s)) == e
            butFirst(insert(e, s)) == s
            last(insert(e, s)) == if s = empty then e else last(s)
            butLast(insert(e, s)) == if s = empty then empty
            else insert(e, butLast(s))
            isEmpty(empty)
            ¬isEmpty(insert(e, s))
            length(empty) == 0
            length(insert(e, s)) == length(s) + 1
```

B LP Input

`Stack.lp` and `Q2.lp` contain the result of translating Stack and Q2 into LP input.

```
%% Stack.lp
execute Sequence_Axioms
set name Stack
declare var
  e: E
  s: C
  s2: C
  ..

declare op
  createPre: ->Bool
  createPost: C ->Bool
  pushPre: ->Bool
  pushPost: C, E, C ->Bool
  popPre: C, C ->Bool
  popPost: C, C ->Bool
  topPre: C, E ->Bool
  topPost: C, E ->Bool
  ..

assert
  createPre = true;
  createPost(s) = (s = empty);
  pushPre = true;
  pushPost(s, e, s2) = (s2 = insert(e,s));
  popPre(s, s2) = (~(isEmpty(s)));
  popPost(s, s2) = (s2 = butFirst(s));
  topPre(s, e) = (~(isEmpty(s)));
  topPost(s, e) = (e = first(s))
  ..
```

```
%% Q2.lp
execute Sequence_Axioms
set name Q2
declare var
  e: E
  q1: C
  q2: C
  ..

declare op
  qEnqPre: C, E, C ->Bool
  qEnqPost: C, E, C ->Bool
  ..

assert
  qEnqPre(q1, e, q2) = (length(q1) < 50);
  qEnqPost(q1, e, q2) = (length(q2) = length(q1) + 1)
  ..
```